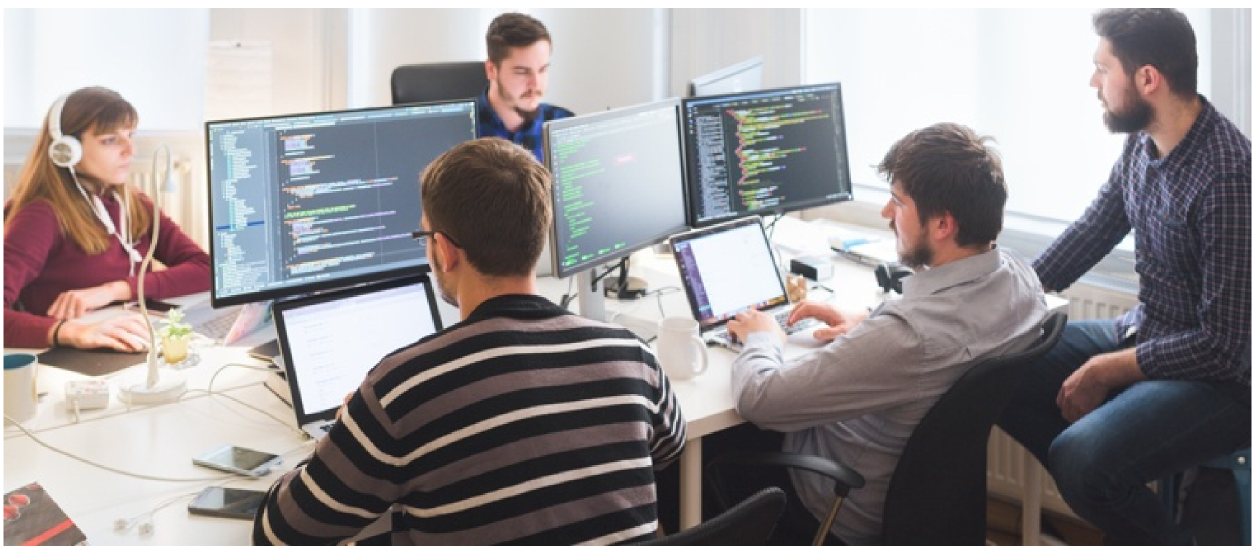
Algoritmos e Estruturas de Dados

Generalized Weighted Job Selection Problem

de ## de 2020

Tomé Lopes Carvalho ------------------------------- 97939 → 33.(3)%
Miguel Beirão Branquinho Oliveira Monteiro - 98157 → 33.(3)%
Gonçalo Fernandes Machado -------------------- 98359 → 33.(3)%



Introdução

Dadas T *programming tasks*, cada uma com *starting date*, *ending date* e *profit* (lucro), e *P* programadores, temos como objetivo encontrar o melhor subconjunto de *tasks* (tarefas), isto é, o subconjunto viável com o valor mais elevado de *profit* total (soma do *profit* de todas as tasks do mesmo).
Para que um subconjunto seja considerado viável é necessário que todas as suas tarefas possam ser feitas pelos P programadores, sendo que cada programador só pode trabalhar numa tarefa de cada vez, e não pode abandoná-la. Uma tarefa também não pode ser feita por múltiplos programadores.

Queremos, então, registar a seguinte informação:
- Lucro máximo do problema;
- Número de subconjuntos de tarefas viáveis;
- Conjunto de tarefas feitas que levam ao lucro máximo (ou um dos conjuntos, caso existam vários);
- Número de subconjuntos de tarefas que levam a cada lucro calculado;
- Tempo de execução da solução;

Descrição do Algoritmo desenvolvido - função *recursive_sol*

Para resolver o problema, desenvolvemos o seguinte algoritmo recursivo que percorre os caminhos de seleção das tasks (forma de árvore binária):

1. Verificar se estamos no caso base. O caso base é quando não há mais tarefas que possam ser feitas (t == T). Significa que encontrámos uma solução viável: incrementamos a variável *n_viable_sol*, registamos a ocorrência do lucro desta solução no *profit_occurrence_arr*, verificamos se este subconjunto é mais lucrativo que o melhor até agora e, caso seja, atualiza-se o *total_profit* e a solução óptima (*opt_sol*). Finalmente, retornamos.

Não estando num caso base, queremos verificar se é mais lucrativo excluir ou incluir a tarefa atual.

2. Para calcular o lucro excluindo a tarefa, vamos chamar recursivamente a função sem atribuir a task a nenhum programador (*problem->task[t].assigned_to = -1* e *recursive_sol(problem, t + 1)*).

3. Verificamos se é possível incluir a tarefa (uma vez que podem estar todos os programadores ocupados no começo da tarefa): sendo *p* o índice do programador que vai realizar a tarefa, utilizamos um ciclo *while* que incrementa *p* até encontrar um programador disponível ou ter percorrido todos os programadores (p == P). Se encontrámos um programador disponível (p < P), vamos alterar o *busy* do programador e o *assigned_to* da tarefa de modo a incluí-la e calcular o lucro, chamando recursivamente a função. Guardamos anteriormente o estados prévios de *busy* e *current_sol_profit* para os restaurar após a chamada recursiva. Como *current_sol[t]* é sempre 0 antes de alterarmos para 1, simplesmente restauramos depois para 0.

Membros adicionados à estrutura *problem_t*

Para a nossa solução, adicionámos 7 membros à estrutura *problem_t*.

Tipo	Nome	Função
int *	opt_sol	Array de 0s e 1s que guarda informação sobre as tarefas a realizar na solução óptima; se *opt_sol[t]* tiver valor 1, a tarefa *t* é realizada na solução óptima.
int *	current_sol	Array de 0s e 1s que guarda informação sobre as tarefas a realizar na solução atual.
int	current_sol_profit	Lucro da solução atual (incrementado ao incluir tarefas).
int	profit_limit	Limite de lucro (soma do lucro de todas as tarefas do problema). É usado como comprimento do array *profit_occurrence_arr*.
int *	profit_occurrence_arr	Array de comprimento *profit_limit* em que os índices correspondem a lucros e os valores ao número de vezes que foram calculados (ou seja, o número de subconjuntos de tarefas que levam a esse lucro).
long unsigned int	n_viable_sol	Número de soluções viáveis. Incrementado cada vez que se chega a um caso base.
char[]	file_name_hist	Nome dos ficheiros utilizados para criar os histogramas (apenas usado para 3 casos).

Função *solution*

Para evitar adicionar código à função *init_problem*, criámos a função *solution* para inicialização dos atributos que adicionámos a *problem_t*.
Nesta função, inicializamos os inteiros *n_viable_sol*, *current_sol_profit*, *total_profit*, *profit_limit* a 0, preenchemos o array *busy* com -1, calculamos o *profit_limit* e inicializamos a zeros o array *current_sol* e alocamos memória para o *profit_occurrence_arr*. De seguida, chamamos a função *recursive_sol(problem, 0)*, para resolver o problema.

Alterações na função *solve*

Para guardar nos ficheiros de *output* informação sobre o lucro máximo e número viável de subconjuntos de tarefas de cada problema, adicionamos as seguintes linhas:

```
fprintf(fp, "Max Profit = %d\n", problem->total_profit);
fprintf(fp, "Number of viable task sets = %lu\n", problem->n_viable_sol);
```

Para os 3 casos escolhidos para representação através do histograma, adicionámos *FILE *hist* (linha 320).
Adicionámos as seguintes linhas para guardar a informação para os histogramas em ficheiros.

```
    hist = fopen(problem->file_name_hist, "w");
    for(i = 0; i <= problem->profit_limit; i++){
        if(problem->profit_occurrence_arr[i] != 0){
            fprintf(hist, "%2d %4d\n",i ,problem->profit_occurrence_arr[i]);
        }
    }

    if (fflush(hist) != 0 || ferror(hist) != 0 || fclose(hist) != 0) {
        fprintf(stderr, "Error while writing data to file %s\n", problem->file_name_hist);
        exit(1);
    }
```

Adicionámos também nas linhas 235 a 240 a seguinte definição, para criar ficheiros com nomes do tipo T_P_ignoreProfits_hist.txt no diretório nMec.

```
#define FILE_NAME_HIST problem->file_name_hist
    if (snprintf(FILE_NAME_HIST, sizeof(FILE_NAME_HIST), "%06d/%02d_%02d_%d_hist.txt", NMec, T, P,
       problem->I) >= sizeof(FILE_NAME_HIST)) {
        fprintf(stderr, "File name too large!\n");
        exit(1);
    }
```

```
#undef FILE_NAME_HIST
```

Método de execução dos casos realizados

Para resolver o problema para os 3 NMECs, para T de 1 a 35, para P de 1 a 8 e com e sem "Ignore profits" ativado, alterámos a função *main* da seguinte forma:

```
int main() {
    problem_t problem;
    int T, P, I;

    int nmec[3] = {97939, 98157, 98359};

    for (int i = 0; i < 3; i++) {               // para cada NMec
       for (T = 1; T <= 35; T++) {              // para cada número de tarefas
           for (P = 1; P <= 8; P++) {           // para cada número de programadores
               for (I = 0; I <= 1; I++) {       // fazer com e sem “ignore profits”
                   init_problem(nmec[i], T, P, I, &problem); // inicializar problema
                   solve(&problem);                          // resolver o problema
               }
           }
       }
    }

    return 0;
}
```

Solução em Java

Uma vez que ainda não estávamos familiarizados com a linguagem C (especialmente com o uso de ponteiros), realizámos primeiro a solução em Java, copiando manualmente um caso de teste (./job_selection 2020 20 4 1).

Antes da função recursiva, fizemos uma função de força bruta, que gerava todos os 2^T subconjuntos de tarefas, verificava a viabilidade de cada subconjunto e, caso fosse viável, calculava o lucro. Não surpreendentemente, esta solução é muito ineficiente, tendo um tempo de execução muito elevado.

```java
public static void brute_force(Problem problem) {
    int n_valid_solutions = 0;
    int maxProfit = 0;
    int col = problem.tasks.length;
    int row = (int) Math.pow(2, col);
    int[][] comb = new int[row][col + 1];   //+1 para o profit
    String format = "%" + col + "s";
    for (int r = 0; r < row; r++) {
        String s = String.format(format, Integer.toBinaryString(r)).replace( oldChar: ' ', newChar: '0');
        for (int c = 0; c < col; c++) {
            comb[r][c] = Integer.parseInt(String.valueOf(s.charAt(c)));
        }
        //inside row
        int profit_if_valid = profit_if_valid(problem, comb[r]);
        comb[r][col] = profit_if_valid;
        if (profit_if_valid != -1) {
            n_valid_solutions++;
        }
        //comb[r][col] = profit_if_valid(problem, comb[r]);
        //System.out.println(s + ": " + comb[r][col]);
        if (comb[r][col] > maxProfit) {
            maxProfit = comb[r][col];
            //System.out.println(s + ": " + comb[r][col]);
        }
    }
    problem.n_viable_sol = n_valid_solutions;
    problem.total_profit = maxProfit;
}
```

Esta solução foi feita de uma maneira pouco ortodoxa: geram-se os números de 0 a 2^T, convertem-se para String em binário. Utilizam-se os caracteres da String para saber quais as tarefas feitas em cada caso (por exemplo, a String “10010000000000000000” significa que são feitas as tarefas 0 e 3). Os caracteres da String são convertidos em inteiros e armazenados na matriz *comb*, em que cada linha representa um subconjunto. *comb* tem T + 1 colunas, sendo esta última coluna utilizada para armazenar o lucro do subconjunto representado pelas restantes colunas da linha. Considera-se o lucro igual a -1 se o subconjunto não for viável.

A função *profit_if_valid* verifica se é possível atribuir um programador a todas as tarefas do subconjunto, retornando -1 caso não seja, ou o *profit* da solução caso seja.

```java
public static int profit_if_valid(Problem problem, int[] solution) {
    int profit = 0;
    for (int i = 0; i < solution.length; i++) {
        if (solution[i] == 1) {
            if (problem.assign_programmer(i)) {
                profit += problem.tasks[i].profit;
            } else {
                profit = -1;
                break;
            }
        }
    }
    //reset ao busy e "assigned_to"s para a próxima vez que for usada a função
    Arrays.fill(problem.busy, -1);
    for (Task t : problem.tasks) {
        t.assigned_to = -1;
    }
    return profit;
}
```

Esta função utiliza outra função, *assign_programmer*, método da class *Problem*, que tenta atribuir ao primeiro programador disponível a tarefa. Caso não haja nenhum programador disponível, retorna *false*.

```java
public boolean assign_programmer(int task_index) {
    if (tasks[task_index].assigned_to == -1) {
        int start = tasks[task_index].start;
        int end = tasks[task_index].end;

        for (int p = 0; p < P; p++) {
            if (busy[p] < start) {
                busy[p] = end;
                tasks[task_index].assigned_to = p;
                return true;
            }
        }
    }
    return false;
}
```

```
"C:\Program Files\Java\jdk-15.0.1\bin\java.exe"
Max Profit: 36148
Number of Viable Solutions: 155510
Execution time: 2584ms
```

```
"C:\Program Files\Java\jdk-15.0.1\bin\java.exe"
Max Profit: 36148
Number of Viable Solutions: 155510
Execution time: 23ms
```

Verificou-se que, para o caso de teste, o tempo de execução da solução de força bruta foi cerca de 112 vezes superior ao tempo de execução da solução recursiva.

Solução recursiva em Java

Até um certo ponto, desenvolvemos a solução recursiva em Java. No entanto, como nos tornámos mais confortáveis com C, acabámos por passar uma versão não final para C e melhorá-la. No final, atualizámos a versão em Java para comparar os tempos de execução. Dado que a versão em Java se assemelha bastante à de C mas com uso de objetos, decidimos não a explicar separadamente.